SAND--90-0030C

DE90 005326

MERLIN: A SUPERGLUE FOR MULTICOMPUTER SYSTEMS

Creve Maples [*]
Sandia National Laboratories
Albuquerque, NM 87185

and

Larry Wittie [+]
Computer Science, SUNY
Stony Brook NY 11794-4400

ABSTRACT

Merlin is a memory based, interconnection system designed to provide very high-performance capability in a distributed multicomputer environment. By using dynamically mapped reflective memory operations, the system creates a virtual memory environment which permits users to utilize both local and shared memory techniques. This mapped virtual memory approach permits selected information to be shared at high speeds and with relatively low latency. There is no software involvement in the actual sharing of information and the system automatically overlaps computation and communication, to the extent possible, on a word-by-word basis. Memory-to-memory mapping allows Merlin to provide a uniform programming environment which is independent of interconnection topology, processing elements, and languages.

1. Introduction

As physical limitations (such as the speed of light) make it progressively more difficult to construct faster and faster supercomputer uniprocessors, interest has begun to focus more heavily on the potential of parallel processing in solving the computational challenges of tomorrow. The rapid advances in microprocessor technology (with 100 MIP processors anticipated within a few years) make the idea of multi-micro systems particularly attractive, both in terms of performance and cost.

Attempts to develop multi-micro architectures have thus far focused primarily on either of two approaches - tightly coupled, shared memory systems, or loosely coupled, message passing systems. Shared memory systems must deal with the difficult problem of memory contention. This problem has been architecturally addressed in several ways: memory interleaving (e.g. Cray X-MP, etc.); memory interconnection networks (e.g. NYU Ultracomputer [1], IBM RP3

[*]This work was performed at Sandia National Laboratories and supported by the U.S. Department of Energy under contract number DE-AC04-76DP00789.

[+]This work has been supported in part by National Science Foundation grants for CER DCS83-19966, research CCR87-13865, equipment CCR87-05079; by National Aeronautics And Space Administration grant NAG-1-249; and by Office of Naval Research grant N00014-88-K-0383.

[2], etc.); and distributed caching (e.g. Sequent, Encore, Alliant, etc.). Combinations of these techniques have also been used to further minimize shared memory contention problems (e.g. Cedar's use of caching and networking [3], IBM 3090's use of interleaving and caching).

Communication, and attendant latency, has been a primary limiting factor in the utilization of message based machines such as the Cosmic Cube [4], N-Cube, Intel iPSC, etc. It should be noted, however, that performance limitations faced by such architectures are not simply due to the communication bandwidth of the system. The software overhead involved in handling messages (e.g. operating system calls, message formatting, transmission protocols, interrupt handling, moving information into and out of buffers, etc.) is frequently the dominating factor in determining system latency [5]. Indeed message based architectures are often only recommended for problems with low interprocessor communication requirements [6].

Some hybrid architectures have attempted to combine the two approaches. The BBN Butterfly, for example, offers a global address space but the physical memories are separate and locally attached to each processor [7]. A delay therefore occurs when a processor addresses any nonlocal memory location. The design of the Denelcor HEP [8] attempted to hide such latency by overlapping the memory access time with the execution of other instruction streams.

Operational shared memory systems have thus far only been able to successfully support a few hundred processors, at most. Message passing systems with thousands of processing elements, are, however, currently available. Because of the relative advantages of each approach, agreement as to which is 'better' (i.e. easier to program, more general, offers the highest performance, etc.) still remain largely unresolved.

Current research in multicomputer systems is primarily focused on extensions of these two basic architectural approaches. At Stanford, for example, research is underway to develop a shared memory system (DASH) utilizing distributed, directoried caching [9]. The Nectar Project at CMU [10] is developing an efficient message based system to support a hetrogenous processing environment which features low-latency, high bandwidth communication and which is scalable. MERLIN (MEmory Routed, Logical Interconnection Network), however, provides a different perspective of processor-memory space.

2. Conceptual and Programming Model

A MERLIN environment pictures the world as consisting a collection of processing elements (not necessarily identical), each of which is independent and has its own local memory. It also assumes the existence of a large, separate (virtual) memory space to which each processor is connected. In SIMD operation, each processor behaves completely independently, and has no effect on the operation of any other processor in the system.
If a user desires to run a problem in parallel, a set of cooperating tasks are loaded onto a selected set of processors (typically allocated by the operating system). During the execution of the problem, various tasks may need to share or exchange information. Some information may need to be shared globally, for example, while other data may only need to be shared with a subset of the processors. To accomplish this sharing, a processor may request that the

system allocate a shared virtual memory region, of the appropriate size, and map the processor's corresponding local memory region into this virtual space.

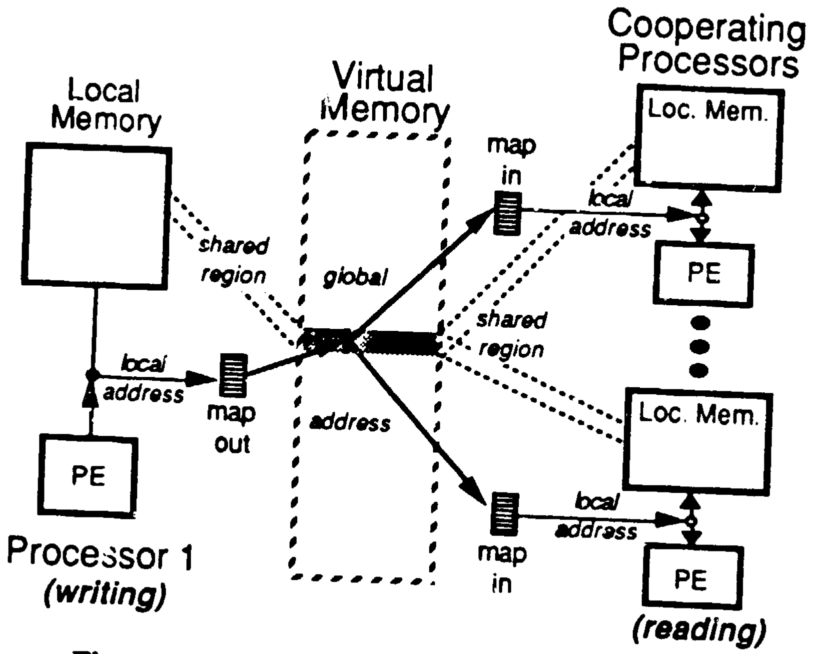


Figure 1 - Programmer's model of shared memory operation in a Merlin system.

Once a global virtual region is allocated, it can, from a programming perspective, be treated as if it were a physically shared memory. While write operations to all other locations are stored locally, writes to a shared region *appear* to be stored in virtual memory. Other processors desiring access to this region (read, write, or read-write) simply have the operating system map an appropriately sized region of their local address space into the desired global virtual region.

The programmer's model of the operation of shared virtual memory is illustrated in Figure 1. In this example, the address of every store operation to processor 1's memory is used as an index to a local (output) mapping table. If an entry is found in the table, the address has been defined as a virtual address, and the table entry is used to translate the address to it's global virtual equivalent. From a programming perspective, the information associated with this address is then stored in global virtual space (see, however, the section on Architecture for what actually occurs). In this conceptual model, the reverse mapping processes would occur when other processors attempt to read a shared address.

It is clear from figure 1 that many different types of shared regions could exist either between cooperative processes working on a single task, or between processes working independently on other tasks in the system. The limitation would simply be on the amount of global virtual address space available. In a multitask environment, a particular process does not have to be executing, or even resident, to utilize shared memory. As long as the shared region exists information can be updated and messages received without active involvement of a process.

Note also that as long as the cooperating processes have agreed on how the shared information is being stored in global virtual memory (e.g. IEEE floating point, 32-bit integer, etc.) the processors on which they execute do not have to be identical (instruction set, cycle time, architecture, etc.). Conceptually, the global memory itself serves as a buffer which permits various components of the system to operate at different speeds.

Thus, in this programming model, a set of processes could elect, at one extreme, to share all data memory and execute in a global memory environment (e.g. Cray X/MP). Tasks operating in such an environment would typically require the use of locks, semaphores, etc., to handle synchronization issues. It is assumed that the virtual memory system will provide support for such traditional synchronization control mechanisms This does not necessarily imply, however, that such capabilities would also exist for a processor's local memory (see also the Synchronization section).

In the opposite extreme, a task could be established which viewed the world as consisting only of local memory processors interconnected by, for example, a hypercube- type of nearest neighbor communication system. For this situation, a separate shared memory region could be created for each processor which spanned only the logically nearest neighbors. Thus when a processor wished to 'send a message' to its neighbors, the information would simply be copied into the shared region. This is not necessarily the most efficient way to accomplish this task on a Merlin system, but it would serve to execute the model, potentially including support for hypercube message routines to permit such code to be run directly.

2.1 Comparison with Models of Other Multicomputer Systems.

Note that the physical communication network in a Merlin environment plays no role in logically programming a hypercube, or any other type of interconnection topology which might be useful to a problem. Indeed one could easily add a global memory region to the previous hypercube example or create other regions for specific purposes (e.g. the logical equivalent of a doubly rooted binary tree for searching).

The use of memory based interconnections eliminate the software involvement and overhead which exists in message passing systems. After a shared region is defined, there is no further software involvement in its use (e.g. system calls, subroutine calls, etc.). Since all shared memory operations involve individual "word packets" (data and address), there are no message protocols involved and no optimal message sizes.

By using local memory as much as possible for storage operations (instruction and data) and utilizing shared memory only in user defined areas, Merlin can reduce memory access conflicts which occur in actual shared memory systems (e.g. Cray X/MP). It's operation is also conceptually differen: than cached based, shared memory architectures. Such systems operate on a *'demand'* basis. If information required by a program c not reside in it's cache, the

its

data must be located in the system and moved into the local cache (with appropriate action being taken to preserve cache coherence in the event that other copies of the same data are also in use). Merlin is an *'anticipatory'* system. By asserting that a region is shared, one is implicitly stating that processes attached to this region will, at some point, require access to the information it contains. Armed with this program supplied, preknowledge, Merlin attempts to reflect any change to information in the region, word-by-word, as fast as it becomes available. In this manner it is anticipating the requirement for the information and may be able to supply it before the actual need exists.

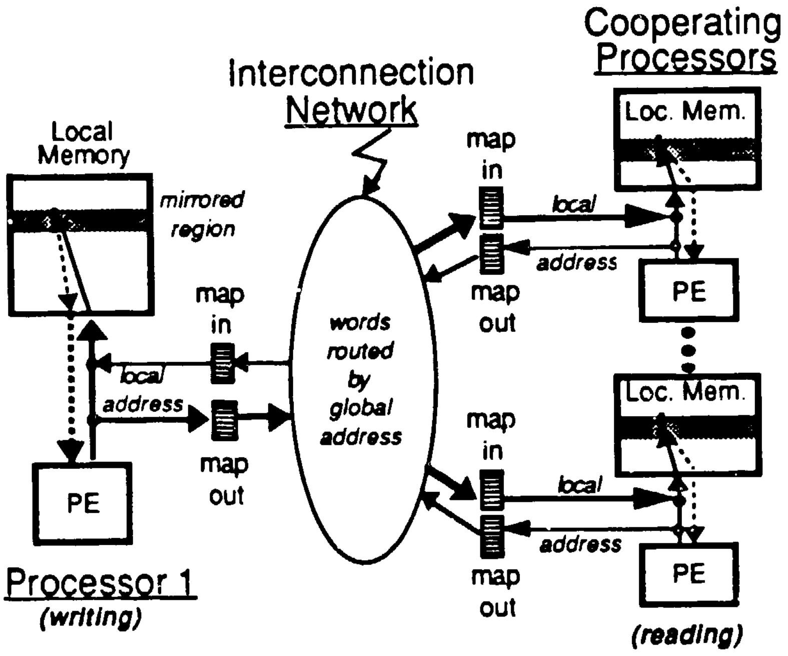


Figure 2 - Memory routing and reflective memory operation in a Merlin system.

3. System Architecture

Developing a conceptual programming model and designing an architecture to realize it, are very different problems. The reality of architectural design, even at a high level, make practical trade-offs inevitable. These, in turn, impact the programming model, usually negatively. Clearly in the conceptual model described previously, a major problem would exist in the implementation of the global memory system. Such a memory would have all the problems of any shared memory system - access conflicts (even though the conceptual model reduced the frequency of shared memory access), potentially high memory latency, physical size limitations - which can significantly reduce the overall performance of the system.

Although useful as a programming concept, the global memory does not need to actually exist. In fact by eliminating it, most of the problems associated with shared memory can be avoided. The global shared memory, illustrated in figure 1, in reality serves as a communication system between sets of processes. It should therefore be possible to replace it with an interconnection network, as shown if figure 2. Such a network would have to transmit "word packets", as opposed to messages or DMA transfers, in order to support the operation of the conceptual model.

3.1 Reflective Memory Operation

Rather than storing shared information in a separate global memory, it can be stored in the physical memory of each processor. As shown in figure 2, this space already exists, by definition. In order to maintain the programming model of a shared memory, any change to a shared location must be *reflected* to all logically equivalent locations in the memory of processors which share this space. Shared regions are thus *duplicated* (as long as they are active) in the physical memory of all associated processors.

This use of reflected or mirrored memory considerably simplifies the architectural design, removes some potential bottlenecks, and generally enhances system performance. As illustrated in figure 2, all write operations to shared regions (solid lines) are <u>both</u> stored locally (as would normally occur) and broadcast, by the interconnection network, to the logically equivalent shared location in the memory of every participating processor. All read operations (dashed lines) occur locally from the processor's own memory (or cache). A simple, non-mapped, reflective memory system, of fixed size, developed by Gould, demonstrated efficient, high performance operation [11].

Replicating shared regions in each processor's memory may appear costly but it is mitigated by three factors: 1) the shared regions and their size is completely controlled by each user program; 2) maintaining multiple copies of the regions increases system performance and scalability (there are no delays or contention problems on shared read operations); and 3) technologically, memory chips will continue to get denser and cheaper. Note that both message based and shared memory architectures (which use distributed caching) maintain multiple copies of data.

A serious problem for all these architectures is latency reduction - if a process requires data which is not available locally, but is available elsewhere, how long is the delay until it is received? In a message based system, this would be the time necessary for the sender to transmit the information to the requester, and may also include the time necessary to notify the sender that the information is needed. In a distributed cache system, the need for the data would be signalled by a cache miss. A correct copy of the information would then be located in the system and the transferred into the requestor's cache In a Merlin system, a request for information is unnecessary. Since the need for the information has been pre-established, if it is available anywhere it will either be present in the processor's local memory or it is already en route.

By dealing with individual "word packets", Merlin further reduces latency by automatically overlapping computation and communication to the fullest extent possible. Every word written to a shared region is broadcast <u>as</u> it is being locally stored. Since the transmission of shared

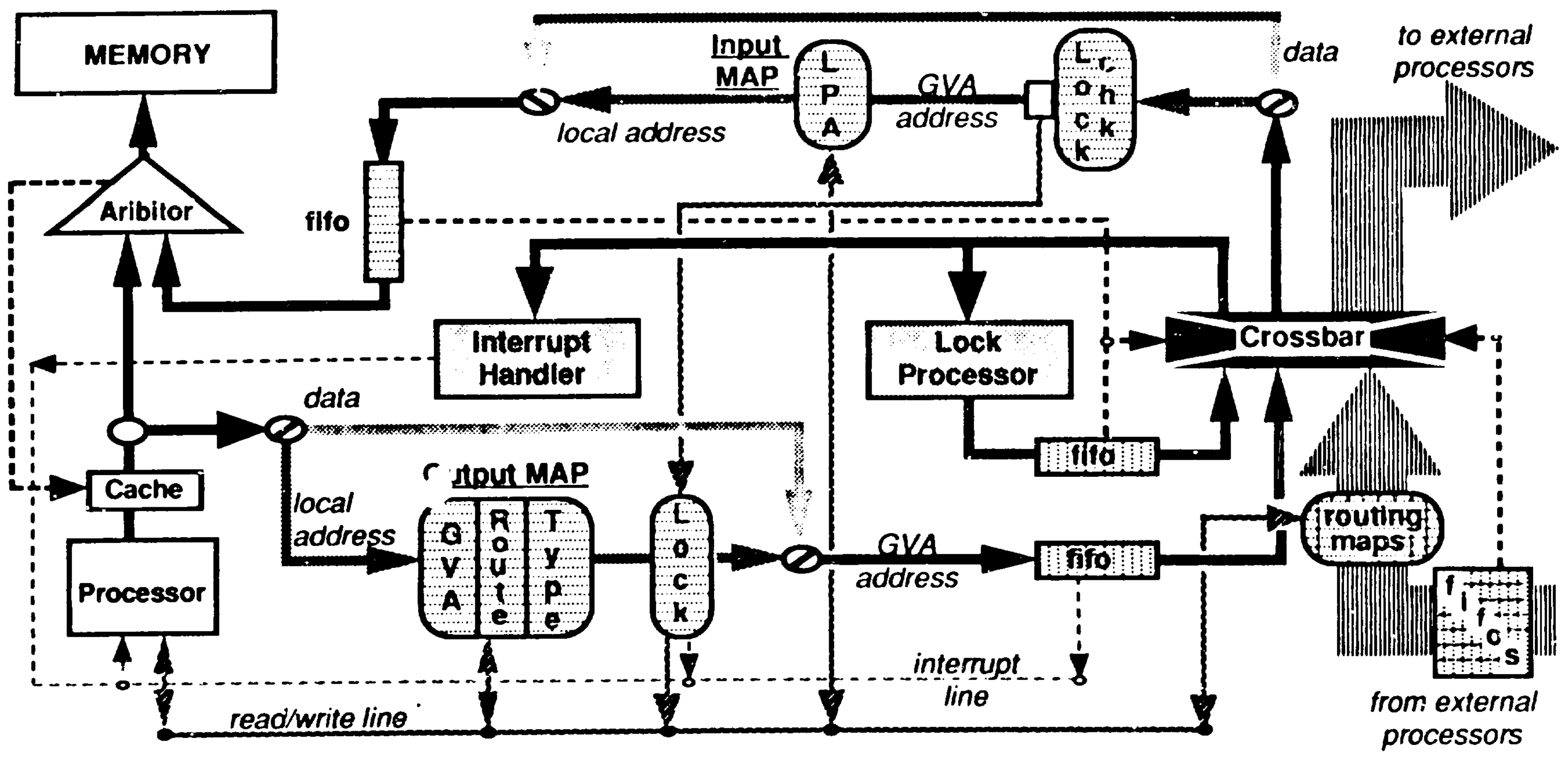


Figure 3 - Conceptual layout of single processor interface to global virtual memory system. Illustrates reflective memory mapping, flow control, routing, and virtual memory management (locks, interrupts, etc.)

data is completely independent of the local processor, it occurs in parallel with the processor's continuing computations (which may, for example, be the computation of the next shared quantity). Neither writing nor reading shared data slows down the operation of a processor, since both operations actually occur normally in the processor's own memory (see also Section 4, on Synchronization).

3.2 Management of Global Virtual Memory

The functional operation of a Merlin network, and the management of the virtual memory system, has been discussed elsewhere [12]. Figure 3 gives a schematic picture of data and control flow in a single processor Merlin interface. In this conceptual representation, the processing element (PE), its memory, and cache (if present) are shown on the left. To interface the PE to a Merlin environment, it's memory bus is "T'd" so that a duplicate copy of all store operations is sent to an independent Merlin communication card.

The communication card has two parts: a processor interface and an interface to the interconnection network. The first part translates local physical addresses (LPA), which are associated with shared regions, into global virtual addresses (GVA), and vice versa. A copy of the PE's memory stores are separated into address and data lines. A masked portion of the LPA is used as the index into a hardware table called the Output Map. Entries in the map are made by the PE when a mapped region is established. If no entry is present for a LPA, no further action is taken and the address (with it's associated data) are ignored.

If an entry is present, the LPA index is replaced with the entry in the table to create the appropriate GVA. Note that this type of remapping operation is very standard and can be carried out efficiently and at high speeds. When the PE initially enters a GVA into the table, it also stores some additional information. These include local routing information and, optionally, data type information. The routing information is used locally to determine the output lines on the communication card to which the global word packet will be routed (see discussion in following section).

The GVA is then used as an index into a second map table to determine if the corresponding global virtual page has been locked If a lock is found, the global write is terminated and an interrupt is generated back to the PE (note that the PE can also read the lock table to determine status directly).

Assuming no lock is present, the GVA, with associated routing and type information, is combined with the original data to form a global 'word packet'. By current design specs, the packet would be about 128-bits long, consisting of 32-bits of virtual address, 64 bits of data, and at least 32- bits of associated data. The additional data would include such information as the ID of PE, hopcount, data type information, and various status bits. The routing information is also associated with, but not part of, the 'word packet', since it will be lost in the next stage. The global word packet then moves to a FIFO buffer for transmission to other PEs in the next stage of operation.

Receiving global information from other processors is essentially the reverse of the above procedure, albeit somewhat simpler. When a word packet, involving data shared by the processor, is received, communication interface routes the information to the input channel of the communication card. As shown in figure 3, the packet is split into address and data lines and the GVA is used as an index into a lock table. If the global page is unlocked, or if the writer has the key, the GVA is passed to the Input Map.

Again the GVA is used as an index into a translation table. If there is no corresponding entry in the table the write is killed, even though it was specifically routed here and passed through the lock check. This is an important since it gives each PE absolute control of it's own environment and security. Only the local PE can place (or remove) entries in it's associated Input Map. Thus no outside operation can access the PE's local memory without prior approval of the PE (by establishing a shared region).

If an entry is present, the GVA is replaced with the corresponding LPA from the table and the address and data lines are merged. The address replacement guarantees that an external store operation is completely confined to the local memory region allocated by the PE. The local address and data is then sent to a FIFO to be stored in the local memory as rapidly as possible.

The manner in which the local memory is physically accessed is, of course, determined by the individual PE architecture. If the memory is dual ported, for example, the external writes would simply access a second port. Alternatively, an arbitration circuit might be necessary to permit the external writes to occur when the memory bus was available (see also Section 3.4, Flow Control). If the PE is a cache based architecture, it would also be necessary to invalidate any cache lines containing the same address.

3.3 Interconnection System

The second stage of a processor's communication card is actually a node in the interconnection system. It should again be stressed that, in order to conform to the conceptual model, the nature of the interconnection (e.g. single bus, multiple bus, point-to-point mesh, etc.) is immaterial to the programming. Although performance may vary, a program written for a Merlin environment will execute on any other Merlin system, even if the interconnection network is totally different

With this caveat, the interconnection system selected for the prototype study is an N-dimensional mesh. Initially this will be a 2-dimensional toroidal mesh so that each PE will be directly connected to four others. Each connections is designed as unidirectional, point-to-point interconnect, so that each communication node would have four external input and four external output channels, coupled in pairs to neighboring processors. The fact that the connectivity may be varied relativity independently of the rest of the system (e.g. to support more traffic or a larger system) is a significant advantage of the Merlin design.

The interconnection channels are illustrated by the large arrows at the right side of figure 3. Each input channel has associated with it a FIFO (for buffering) and a routing map. The routing maps are tables, indexed by GVAs, which contain a bit mask defining the local routing path for the corresponding GVA. Assume, for example, there are six exit channels (two internal and 4 external channels, as shown). Each entry in the routing table would then be 6 bits long, with each bit corresponding to a exit channel. An entry of 001111 would therefore indicate that the global word packet should be broadcast on all four of the external channels.

Note that the path of a global data word is determined by its GVA and the entry in the routing table of each node through which it passes. It is not carried with the data. The path or virtual circuit between any two processors sharing the same virtual address space is therefore established by the system at the time the shared region is created. Under normal circumstances (baring system problems), these paths remain fixed for the life of the shared region. The paths are created by the PEs entering the necessary routing information in the routing table corresponding to the appropriate incoming channel.

In order to be able to maintain a weak form of data consistency, it is important that the interconnection system guarantee that *the order in which information is stored by any processor will be the order in which it is received by all participating processors.*

Other processors can easily be added to an established shared region by simply creating a path from the nearest PE which already contains the region. Similarly, failed or otherwise occupied processors can simply be routed around without user involvement. This capability also has important debugging applications. A shared region may be mapped into a previously uninvolved PE for debugging purposes. If selected carefully, the presence of this debugging PE will not change the characteristics of the system under study, and can be used to perform realtime analysis of shared memory activity.

3.4 Flow Control

There are three units which govern the dynamics of information flow in the system: the crossbar switches (one per PE); FIFO buffers (~7 per PE), and the PEs themselves. The crossbar, shown in figure 3, is designed as a round robin, multipass switch. Essentially this means that each of N input channels is guaranteed access to the switch every Nth cycle. Within each cycle, however, any output channels not needed by the primary requester will be utilized to satisfy the requests of other active channels. The only requirement is that no input channel will be permitted switch access unless all of it's output needs can be met simultaneously (i.e. no partial firing).

Because of the dynamics of the switch and the system itself, it is obvious that buffers must be available on each communication channel to facilitate flow control and to handle transient hot spots. Figure 3 shows a number of FIFO buffers in the layout of each PE's communication card. Essentially there is a separate buffer on each input line to the crossbar switch and one to buffer external writes to the PE's local memory. In a system connected as a 2-D toroidal mesh, there would be 7 buffers associated with each PE. The optimal size of these buffers are currently being investigated by simulations, but is probably on the order of 1K words each.

Although the FIFO buffers should be able to smooth out normal dynamics of the system and handle transients, it is possible that more serious hot spots could develop (due perhaps to inappropriate routing) or that the system could be over driven by a particular program. Figure 3 illustrates that every FIFO is equipped with a status line signalling a half full condition. All but one of these lines are connected to the crossbar switch's control logic. In the case of the two internal, output channels (to the Input Map and Lock Processor) these status lines will inhibit the crossbar's control unit from permitting access to these channels until the half full signal is removed.

The FIFOs associated with externally connected channels operate a little differently. When the FIFO of any channel signals the crossbar switch, the switch immediately sends an inhibit signal on the external line connected to the same PE which is transmitting to the half full buffer. This signal is stripped off by the transmitting processor's receiver (not shown) and immediately sent to the crossbar control unit on that PE. This signal inhibits the sender's crossbar from transmitting more data over the congested channel until further notice (i.e. it's FIFO is less than half full).

The system, thus far, has gracefully handled the heavy loading, filling FIFOs, temporarily closing communication channels, and generally allowing the problem to propogate, from node to node, to buffers downstream, and ultimately to the source(s). No PE has actually yet been effected, and, if the activity is transitory (i.e. burst), equilibrium will automatically be reestablished without any PE involvement.

At some point, if the heavy traffic continues, a crossbar switch somewhere in the network will be forced to block the input line from it's local, transmitting PE (see figure 3). When this happens, the FIFO associated with this line will begin to fill. When it reaches half full it will send an inter- rupt to the local PE itself. This interrupt signal will cause acontext switch to the operating system, which will stop the active process from further transmissions. The PE's operating system may respond to such an interrupt in a variety of ways: wait until the condition has resolved itself (i.e. the interrupt goes away); execute a different task; investigate the problem (via communication with other PE's over an independent ethernet connection); notify (if the software exists) the transmitting task; notify the user; or, in bad situations, move processes to different PEs or attempt to reroute the virtual circuit. The system managment issues in such situations are the subject of continuing research.

Simulations of deliberately over driven systems have shown that the hot spots tend to occur at the interface to a PE's memory (i.e. a memory bandwidth limitation) and not within the network. This simply implys that, under worst case conditions, one PE is capable of writing information faster than another PE can absorb it. While it is hoped that such situations will be avoided by users, it is important to understand that they are not necessarily bad.

It is not unusual in multiprocessing systems, both message based and shared memory, for a PE to be temporarily blocked while access to information is being acquired. In a Merlin system, even if a PE is interrupted, the transmission of data and it's storage in memory is occurring in parallel, as rapidly as possible, throughout the network. To look at the problem differently, if a user rapidly generates large amounts of shared data, the performance of the system may be limited by the memory bandwidths of the PEs. A Merlin system would transfer data to all the processes at memory bandwidth speeds, which, given the basic problem definition, is as efficiently as it can be executed.

4. Synchronization

Synchronization is basically an algorithmic problem, not an architectural one. It is also true that a particular parallel architecture can make synchronization difficult and thereby significantly slow down the performance of the system. The need for synchronization, however, is a result of the approach taken to solve the problem. The requirement for frequent synchronization will adversely effect the performance of a problem on any parallel architecture. Merlin offers a number of methods of dealing with synchronization requirements, ranging from self-synchronized messages, to global memory locks, to semaphores, to test-and-set operations, to the technique of phased synchronization.

4.1 Locks, Semaphores, etc.

Since global virtual memory does not exist, it's size is only limited by the size of the address lines in the network. In the Merlin prototype currently under design, this is 32-bits, with provisions for expansion. To facilitate inter-system communication and global memory management, the operating system reserves, at boot time, several pages of the global virtual address space for each PE in the network.

One such region is utilized for OS to OS communication. At start up time the system creates virtual circuits (routing paths) between each PE and the reserved OS page of every other PE (i.e. it creates a spanning tree rooted on each PE and covering all nodes). In a similar fashion, other dedicated regions are established on each PE to handle interprocessor interrupts, memory locks, and fetch and add operations.

Figure 3 shows a special routing circuit on each communication card for handling such operations. This consists of an interrupt handler, a lock processor, and a fifo-buffered return channel to the crossbar switch. One PE can interrupt another, for example, by writing to the GVA reserved for that PE's interrupt region This write is routed to the interrupt handler on the selected PE, which in turn actually interrupts the processor (and also supplies the ID of the interrupting PE, and a 64 bit status word).

Memory locks are handled in a somewhat similar fashion. If a region requires a lock, a request to establish one must be made to the system, and a lock address (and lockmaster) is allocated. To acquire a lock, a PE simply writes to the appropriate global virtual lock address. The lock handler on the lockmaster processes the request and returns the result to the sender. If the request is granted, a write is first initiated to the global region involved (and hence to all PEs sharing this region). Special status bits are set in this write which are intercepted by the lock checker (figure 3) on each associated PE. The lock checker recognizes this write as a 'lock set', marks the region locked (for both input and output), and stores the 'key' or ID of the lock owner. The local PE is not involved in any of these operations.

Each lock processor is also designed to support a primitive fetch-and-add type of capability within it's own special memory. A fetch-and-add location is requested and assigned by the system. A variable can then be stored in the location. Writes to this location by any PE will cause the lock processor to return the current value of the variable to the PE, and add the value written to the variable. This is carried out in one cycle and useful for such functions as queue management, etc.

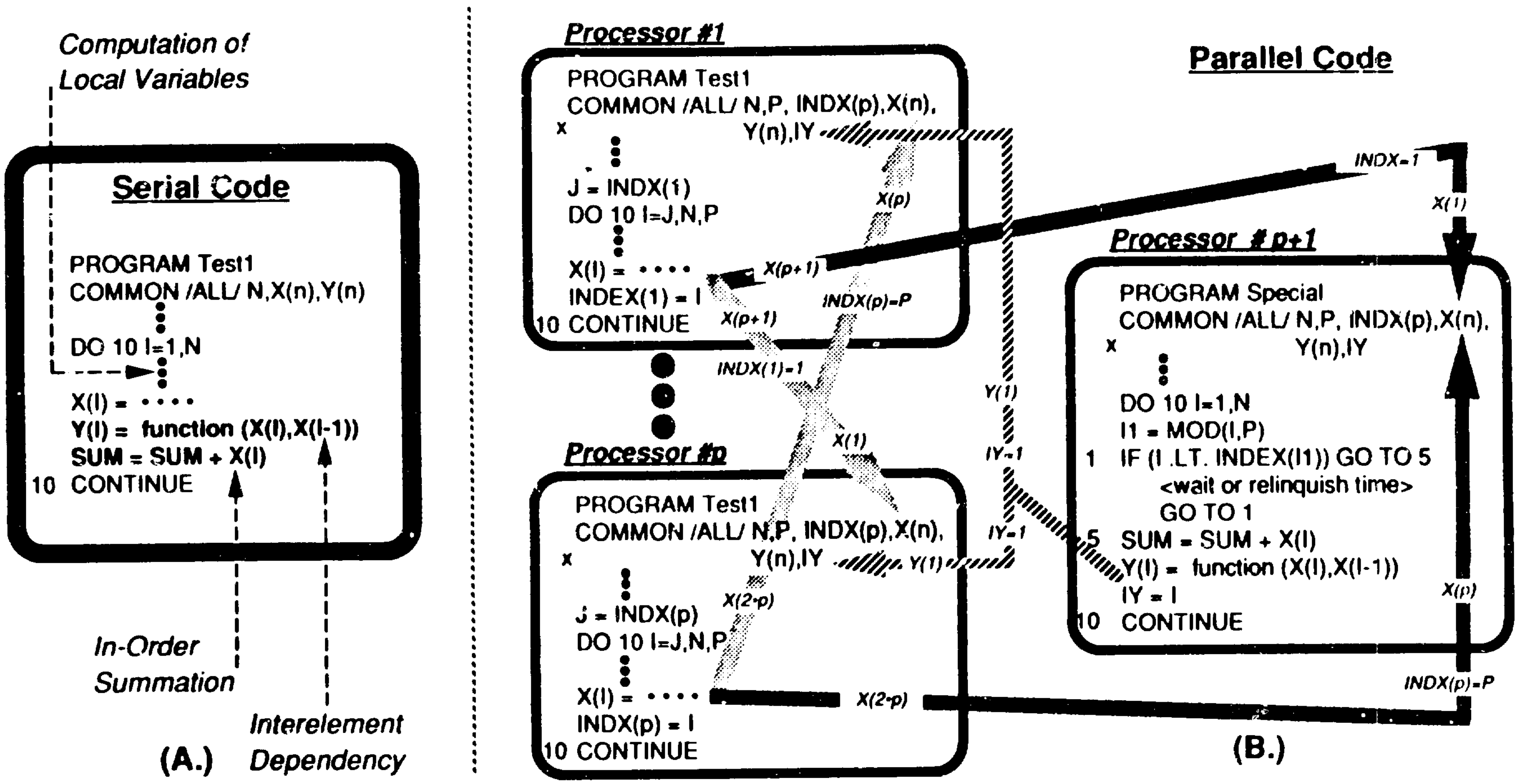
4.2 Phased Synchronization

Phased synchronization is an method of handling interdependent parallel operations in an quasi-asynchronous, overlapped fashion. The approach is often quite successful in eliminating synchronization barriers and other bottlenecks to efficient parallel operation [13]. A limited type of phased synchronization was supported in hardware on the Alliant Computer FX/8 [14]. In this case, interdependent loop computations were identified by the compiler and the associated memory references tagged. Processors could then execute independent loop segments until a tagged, or dependent location was encountered. The process was then halted until the necessary value was stored by another processor.

The approach taken by Alliant, while serving as an example, was restrictive and not scalable. Generally speaking, for phased synchronization to be practical it is necessary that the supporting architecture be capable of frequently broadcasting small amounts of information throughout the system with minimal overhead and latency. This can be illustrated by the simple example in figure 4.

Figure 4A shows a small serial fortran code with a loop containing an interelement dependency (y(i+1) cannot be computed until x(i) is calculated) and requiring the computation of an in-order summation (perhaps for reproducibility due to possible round-off errors). Such a program would typically not be amenable to parallel processing. Figure 4B, however, illustrates how the problem can be run in parallel by utilizing a phased synchronization approach.

The program is first separated into independent and dependent phases. The dependent code, shown on the right in figure 4B, contains the essentially serial portion of the computation. The remaining independent code is, by construction, now parallelizable. The 'N' executions of the loop is now spread over 'P' processors as shown. In order share the results of the parallel computation, a shared common block called 'ALL' is created, containing the arrays X and Y where the computational results are stored. In this example the region ALL is defined as a globally shared region prior to the execution of the programs. Figure 4B illustrates how computational results are then simultaneously transmitted and stored, word-at-a-time, in the local memories of the processors.



```
PROGRAM Test1
COMMON /ALL/ N,X(n),Y(n)
   :
DO 10 I=1,N
   :
X(I) = ....
Y(I) = function (X(I),X(I-1))
SUM = SUM + X(I)
10 CONTINUE
```

```
PROGRAM Test1
COMMON /ALL/ N,P, INDX(p),X(n),
x                  Y(n),IY
   :
J = INDX(1)
DO 10 I=J,N,P
   :
X(I) = ....
INDEX(1) = I
10 CONTINUE
```

```
PROGRAM Test1
COMMON /ALL/ N,P, INDX(p),X(n),
x                  Y(n),IY
   :
J = INDX(p)
DO 10 I=J,N,P
   :
X(I) = ....
INDX(p) = I
10 CONTINUE
```

```
PROGRAM Special
COMMON /ALL/ N,P, INDX(p),X(n),
x                  Y(n),IY
   :
DO 10 I=1,N
I1 = MOD(I,P)
1  IF (I .LT. INDEX(I1)) GO TO 5
      <wait or relinquish time>
      GO TO 1
5  SUM = SUM + X(I)
   Y(I) = function (X(I),X(I-1))
   IY = I
10 CONTINUE
```

Figure 4 - (A.) An example of serial fortran code with a barrier sum and interelement loop dependencies; (B.) A parallel decomposition of the code using phased synchronization.

The dependent computation can not, however, proceed until the necessary information is available from other processors (not unlike a macroscopic form of data flow). To provide this information, an array named 'INDX' was created and added to the globally shared region. Each entry in this array corresponds to a value for the loop index 'I' in each of the P processors. Note that the processors store the current value of this index *after* computing each value of X(I). As shown in figure 4B, the processor executing the dependent code is continually receiving values of X and INDX from the parallel computations.

Since the dependent code needs to execute the loop sequentially, it is necessary to test to verify the presence of the next X value in its local memory or to wait for its arrival. This process is thus executing in a phased manner, synchronizing at each step with the parallel computation. Even though the dependent process can not begin execution until after X(1) is available and cannot end until X(N) is received, most of the serial computation is overlapped with the parallel computation and executed concurrently

This approach requires no formal locks or messages (in the traditional sense), is extremely efficient, and can easily be adapted to handle a wide variety of other situations in which parallel processing is difficult [13] (e.g. conditional branches, load balancing, queuing, etc.). Phased synchronization operates on the principal that if processes can determine, from moment to moment, the status of information in the system, dependent computations can often be safely overlapped with other computations, thus breaking many traditional parallel processing barriers.

5. Summary

Merlin provides a very high performance 'anticipatory' (as opposed to a 'demand') approach to sharing memory in multicomputer environments. It provides a programming model which supports both local and dynamically created shared memory. The system eliminates all software involvement in the actual sharing of information, and yet permits shared memory regions to be created between between any set of cooperating processes. Communication latency between processors is minimized, and essentially eliminated, due to the automatic instruction-level overlap of communication and computation.

The Merlin approach can also be used, in the future, to support heterogeneous multicomputer systems (e.g. micros, mainframes, vector, systolic, database, AI, etc.). It would permit processors either to be utilized independently or, as needed, to be welded together into cohesive subsystems.

The programming flexibility of the system, it's utilization of high-level languages (including mixed language support), and the topology-independent nature of it's operation, should provide users with a versatile, high-performance system that can be logically programmed to match individual application requirements. The ability of a system to rapidly and efficiently disseminate single words of information, permit the use of phased synchronization techniques which, in turn, open up new approaches to parallel problem decomposition in a multiprocessor environment.

Investigations into system design issues are continuing. Simulations are being used to examine routing algorithms and to study network traffic patterns and behavior under heavy loading. Operating system and language extensions, to support the operation of global virtual memory, are also being developed, together with parallel debugging tools.

REFERENCES

[1.] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Randolh, M. Snir, P. J. Teller and J. Wilson, "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach", *IEEE Proc. 12th Intl. Symposium on Computer Arch.*, Boston, MA, June 1985, 126-135.

[2.] G. F. Pfister, W. C. Brantley and D. A. George, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. 1985 Int. Conf., on Parallel Processing*, 1985, 764-771

[3.] D. J. Kuck, E. S. Davidson, D. H. Lawrie and Ahmed H. Sameh, "Parallel Supercomputing Today and the Cedar Project", *Science*, 231, (28 Feb, 1986), 967-974.

[4.] C. L. Seitz, "The Cosmic Cube", *Commun. Assoc. Computing Mach.*, 28, 1985, 22-33.

[5.] L.-Felipe, E. Hunter, M. J. Karels and D. A. Mosher, "User-Process Communication Performance in Networks of Computers", *IEEE Transactions on Software Engineering 14*, 1, (January 1988), 38-53.

[6.] Geoffrey C. Fox, "Concurrent Processing for Scientific Calculations", *COMPCON 84*, San Francisco, CA, February, 1984, 70-73.

[7.] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken and T. Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor", *Proc. of the Int. Conf. on Parallel Processing*, August, 1985, 531-540.

[8.] B. J. Smith, "A Pipelined Shared Resource MIMD Computer", *Proc. 1978 Int. Conf. on Parallel Processing*, 1978, 6-8

[9.] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence", *15th Intl. Symposium on Computer Architecture*, 1988.

[10] E. A. Arnould, J. Bitz, E. C. Copper, H. T. Kung, R. D. Sansom and P. A. Steenkiste, "The Design of Nectar: A Network Backplane for Hetrogeneous Multicomputers", *Proc. 3rd Int. Conf. Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS III)*, Boston, MA, April, 1989.

[11.] Christopher Wilks, "SCI-Clone/32 - A Distributed Real Time Simulation System", *Computing in High Energy Physics*, edited by Hertzberger and Hoogland, North-Holland Press, 1986, 416-422.

[12.] Larry D. Wittie and Creve Maples, "Merlin: Massively Parallel Hetrogeneous Computing" *Proc. 1989 Int. Conf. on Parallel Processing*, St. Charles, Illinois, August 1989, 142-150.

[13.] Creve Maples, "Phased Synchronization and Parallel Computing", to be published.

[14.] Robert Perron and Craig Monday, "Architecture of the Alliant FX/8", *Digest of Papers for COMPCON '86*, March, 1986, 390-393.